Enhanced Maintenance and Explanation of Expert Systems Through Explicit Models of Their Development

Enhanced Maintenance and Explanation of Expert Systems Through Explicit Models of Their Development

ROBERT NECHES, WILLIAM R. SWARTOUT, AND JOHANNA D. MOORE

Abstract—**Principled development techniques could greatly enhance the understandability of expert systems for both users and system developers. Current systems have limited explanatory capabilities and present maintenance problems because of a failure to explicitly represent the knowledge and reasoning that went into their design. This paper describes a paradigm for constructing expert systems which attempts to identify that tacit knowledge, provide means for capturing it in the knowledge bases of expert systems, and apply it towards more perspicuous machine-generated explanations and more consistent and maintainable system organization.**

Index Terms—**Expert systems, explanation, natural language generation, software development, software maintenance.**

I. Introduction

SWARTOUT'S XPLAIN system [1] demonstrated the feasibility of producing expert systems with enhanced capabilities for generating explanations and justifications of their behavior. XPLAIN was based on two key principles: explicitly distinguishing different forms of domain knowledge present in the knowledge base, and formal recording of the system development process. We will argue that these principles are vital both for explaining and for maintaining expert systems. This paper will propose a new paradigm for building expert systems, and consider the paradigm's implications for providing automated assistance in two tasks commonly encountered in the course of developing and using expert systems:

- generating explanations to clarify or justify the behavior and conclusions of the system;
- extending or modifying the system's knowledge base or capabilities.

The paradigm we are proposing, which we call the *Explainable Expert Systems* approach, calls for shifting the emphasis of knowledge engineers' efforts from procedural encoding to declarative knowledge representation. In this approach, development and use take place in an integrated support environment. Knowledge engineers and domain experts collaborate to produce a rich semantic model of the declarative and procedural knowledge of the domain. Their efforts produce a knowledge base which, augmented by advice about implementation considerations, is used to guide an automatic program writer through generation of the actual code for the expert system. The program writer maintains a record of its choice points and decisions, which constitutes the system's *development history*. The code is executed by an interpreter that maintains a record of the system's *execution history*.

All together, the knowledge base, the development history, the code, and the execution history, provide the basis for question-answering routines that allow developers and users to obtain information about the origins and rationales behind the system's code, as well as about the code itself. The availability of this extra information provides domain experts with more power to critique the system, to identify deficiencies, and to find those points in the system conception or implementation responsible for a deficiency. The availability of this extra information also provides end-users greater power to understand the abilities and limitations of the system. Thus, although the *EES* paradigm does increase initial development costs, we expect that those costs will be repaid in greater acceptance by users, easier maintenance, and a greater degree of reusability that will smooth the development of related systems later on. To support this claim, let us briefly consider what the availability of extra knowledge means for the tasks mentioned earlier: explanation and maintenance.

A. Explanation

Consider the following dialog, which occurred during an experiment in which we had human experts play the role of an expert system that would give Interlisp programmers advice and assistance in improving the style of Zetalisp programs. In this dialog, the typed messages exchanged between the expert and the user are in boldface, and their verbal remarks are in italics:[1]

System: **Your file begins with a number of DEFVAR's. Some of these can be changed to DEFCONST. Would you like the system to do this?**

Manuscript received March 29, 1985; revised July 3, 1985. This work was supported by the Defense Advanced Research Projects Agency under Grant MDA 903-81-C-0335. Portions of this paper were presented at the IEEE Workshop on Principles of Knowledge-Based Systems, December 1984, and at the International Joint Conference on Artificial Intelligence, August 1985.

The authors are with the Information Sciences Institute, University of Southern California, Marina del Rey, CA 90292.

[1]The discussion concerns constructs for declaring initial values for global program variables. Interlisp has only one such construct, "DEFVAR." Zetalisp has two, "DEFVAR" and "DEFCONST"; the distinction between them hinges on whether or not the variable's value is allowed to change from its initial value during program execution. The expert has inspected the user's program and found a number of places where he would suggest switching to the latter construct, but the user does not fully understand and trust the advice.

User: *DEFCONST? That advice is ...*
Many of the variables I use are used freely. Should these really be constants?

System: **You should use constants if you do not intend to change the value.**

User: *Hunh? What's that? ... I don't know if it can do that for me. See, now, it said some of these may be changed to DEFCONST, and maybe he's—the answer to this question is he already knows which ones should be constants. Or, maybe it's not like Interlisp in that he can reset constants or something like that.*

The final verbal remarks illustrate a common phenomenon: the user is not sure what the system's capabilities really are, and is busily spinning off myriad hypotheses. Notable in this brief interaction is the number of questions that the user would benefit from being able to ask the system.

- *How general is this advice?*
- *What is its basis?*
- *Why is it desirable?*
- *What does "some of these may be changed" really mean?*
- *How does the construct being suggested differ from the one used in the program?*
- *Does the system know which declarations should be changed?*
- *If not, does it have a way of deciding?*
- *What method would it use?*

Answering such questions depends on knowledge above and beyond that required to generate the initial recommendations—in particular, knowledge about the structure and capabilities of the system, and about the concerns in the knowledge domain that motivated their design. The answers are not merely of tutorial value to the user. They indicate the range of the system's capabilities and the reliability of its conclusions. They affect the user's trust in the system, and help in deciding how best to make use of the system's capabilities. If users can only obtain these answers by consulting human domain experts or knowledge engineers, then the expert system has failed in a fundamental goal: reducing demands on scarce human resources by providing a repository for knowledge and a vehicle for disseminating it.

As we will elaborate in Section IV, the availability of information from a domain knowledge base, a development history, and an execution trace yields the opportunity to provide richer explanations of the system than are avail able in conventional approaches. Conventional expert systems, lacking these added knowledge sources, are restricted to explanations composed from canned text or by paraphrasing the system code. These suffer from a number of flaws. Canned text cannot anticipate, or adapt to, all possible needs. Since its maintenance is a separate **and additional** task from code maintenance, the text can quickly become invalid with respect to the true state of the system code.

On the other hand, code paraphrasing is limited by the information that is represented in the code—and even more limited by the information that is not represented. Explanation by code paraphrasing can describe actions in fairly low-level terms but it cannot, for example, describe high-level principles motivating those actions or explain why those actions instantiate some high-level principle. For example, in MYCIN, the general principle that the type of an infection may be determined using a weight-of-evidence scheme is encoded in several dozen rules, specific to particular types of infections [2]. The general heuristic itself is never explicitly represented, and hence is not available for explanation.

More sophisticated explanations require that the design knowledge underlying an expert system be explicitly represented, which is one of the roles served by a richer domain knowledge base and a development history. Our approach to explanation depends on a taxonomy of question types, with *explanation strategies* associated with each question type. An explanation strategy tells the system how to inspect the knowledge base in order to obtain information relevant to answering a particular kind of question.

B. Maintenance

As we will elaborate in Section III-C, the use of an automatic program writer to derive code from more abstract specifications presents an opportunity to simplify the maintenance process. The need to modify a system's code generally arises for one of three reasons:

- there is an invalid assumption or principle upon which the code was based;
- an assumption or principle was valid, but the code instantiated it incorrectly or incompletely;
- additional concerns, such as ease of implementation or efficiency considerations, make an alternative method of achieving some goal preferable.

In all of these cases, the primary tasks of a maintainer are to diagnose the cause of dissatisfaction with the current system, and to locate and modify all of the relevant code. In conventional systems, since the linkage between code and higher-level principles is not explicitly represented, there can be difficulties with both tasks. When the basis for some segment of code is either invalid or inappropriately realized, it may be very hard to reconstruct from the code alone what that basis should have been. When code is rewritten or superseded, it may be very difficult to determine what other code is affected. Consider the same example mentioned above of MYCIN's weight-of-evidence scheme for determining infection types. If one wished to change this principle, dozens of rules would have to be located and modified. In the absence of any kind of pointers to those rules, it is easy to imagine some of those rules being missed, requiring multiple iterations of modification and testing to accomplish the modification.

Similar issues arise when the goal is to extend a system. Say one wanted to add knowledge about a new infection type to MYCIN. Obviously, the many rules pertaining to

existing infection types could be used as examples to indicate the form of the new rules that would have to be added. Again, though, in the absence of pointers into the code, one has no easy way of making sure that all the relevant rules are located. Thus, one has no assistance in ensuring that all the necessary new rules are added, much less that they are correctly stated.

Our approach is to provide support for the diagnosis phase of maintenance through the extended explanation capabilities, and for the code modification phase through the automatic program writer. As Sections III-A and III-C will show, the approach calls for the system builders to provide a knowledge base containing descriptive knowledge of how the domain works, and abstract problem-solving methods that apply to classes of problems. A *classifier* [3] identifies all instances of domain concepts for which a problem-solving method must be instantiated, and the automatic program writer generates code by integrating descriptive domain knowledge and problem-solving methods. Thus, for example, MYCIN's weight-of-evidence scheme for infection types would be handled by providing a general principle for performing weight-of-evidence evaluations together with the empirical associations between findings and infection types and (possibly) a method for integrating the results of individual evaluations. The program writer would use this information to generate the appropriate specific rules for each particular infection type. Changing the principle or adding a new infection type both would be a matter of changing a small number of assertions in the knowledge base and then rerunning the program writer.

II. XPLAIN: The Precursor of the EES Paradigm

The XPLAIN system recognized two forms of domain knowledge (domain descriptive information versus problem solving methods) and one kind of development (refinement by a hierarchical planner). The domain descriptive knowledge told XPLAIN how the domain worked. In a medical system, the domain descriptive knowledge would typically include knowledge of various diseases and physiological states and causal relations among them. The problem solving knowledge told XPLAIN how to employ its domain descriptive knowledge to achieve particular goals such as disease diagnosis or therapy administration. For example, when XPLAIN was used to generate a digitalis drug dosage advisor, its domain descriptive knowledge (or "*domain model*") included assertions such as the following.

- High serum calcium levels can cause increased automaticity.
- Low serum potassium levels can cause increased automaticity.
- High digitalis doses can cause increased automaticity.
- Increased automaticity can cause ventricular fibrillation.
- Ventricular fibrillation is a highly dangerous condition.

This descriptive knowledge was augmented by problem solving methods (or "*domain principles*"[2]). Domain principles had three major parts: a goal, a prototype method, and a domain rationale. The goal of a domain principle stated what its prototype method could accomplish. The prototype method was expressed in lower-level, more specific terms. When the automatic programmer tried to refine a goal it examined the domain principles to find one whose goal most closely matched the goal to be refined. As an example, the domain principle for compensating for drug sensitivities had a method that stated:

check for drug sensitivites, and if they exist reduce the drug dosage.

Before this method could be further refined, it was necessary to know what the particular sensitivites were for this domain. The *domain rationale* associated with the principle provided a definition of what the term sensitivity meant, e.g., a factor that can cause something dangerous that also can be caused by administering the drug.

By matching this definition against the descriptive domain knowledge, the system could find out what were the particular sensitivities for the domain of digitalis therapy. The domain rationale is the major feature that distinguished the XPLAIN approach from other refinement-based systems [4]. Its purpose was to integrate knowledge from the descriptive domain model into the refinement process by defining terms at one level of refinement using terms at the next level down.

To elaborate, refinement-based systems move through different levels of language: they refine a high-level description of a goal or problem into a low-level one. Yet, the steps between levels of language are often quite implicit. In particular, the correspondence between terms used at one level of language and their realization at the next level down is usually implicit. The domain rationale allowed us to indicate that correspondence.

Applying this principle (and, of course, others) to the descriptive knowledge led XPLAIN to generate procedures for adjusting dosage recommendations to account for serum calcium and serum potassium levels. As it generated that implementation from the two forms of knowledge, XPLAIN recorded the steps it had taken.

Recording the derivation of the actual low-level procedures from the domain principles enabled XPLAIN-generated systems to give more principled answers to "why" questions. XPLAIN's digitalis drug dosage advisor, for example, was capable of explaining that it was asking about the patient's serum calcium level as part of adjusting the recommended dosage, *and that this was important because too high a dosage of digitalis could interact with the effects of serum calcium level to produce the dangerous condition of ventricular fibrillation*. That is, the XPLAIN-generated system could justify its request for a patient parameter both by paraphrasing the program code, and by constructing a justification for the parameter's significance based on an abstract model of the domain. This should be contrasted with conventional expert systems,

[2]The term "domain principles" is a bit misleading. While some of the domain principles were quite domain specific, others (such as principles that set up backward chaining control for assessing symptoms) were largely domain independent.

where only the knowledge needed to perform the task is present. A digitalis advisor built with only performance knowledge would know how to check serum calcium and reduce the digitalis dosage, but would not know—and therefore could not explain—*why* it was doing so. XPLAIN's explanation is clearly richer.

The separation of knowledge in XPLAIN also seemed to hold promise for easing the process of extending the system. Knowledge was modularized into 1) situations where patient factors could have undesirable interactions with the digitalis dosage; and 2) problem solving knowledge governing checking for such factors and adjusting the dosage accordingly. Since XPLAIN took responsibility for applying the problem solving knowledge to whatever domain descriptive knowledge was given to it, programming the system to handle a new situation and generate suitable explanations for its new behaviors would require making only a few assertions to describe the added factors, rather than writing large amounts of new code that bore great similarities to existing code. Unfortunately, this possibility was not explored in any great depth in the XPLAIN work.

A. Limitations of XPLAIN

Although XPLAIN demonstrated some of the promise of this paradigm for expert system development, it suffered from several important limitations in the representation of terminology, in the power of the program writer, and in explanation production.

Although a central feature of XPLAIN was the ability to use the domain rationale to indicate the correspondences between terms at different levels of refinement, in fact, terminology was represented poorly in XPLAIN. No explicit definitions were given. The particular term that a domain rationale defined really depended on what context the program writer interpreted it in. An additional problem was that because XPLAIN's (implicit) terminological definitions were provided by the domain rationale part of domain principles, that meant that the definitions of terms were associated with particular pieces of problem solving knowledge. That does not seem to be appropriate. Instead, terminological knowledge should be shared *across* individual domain principles.

XPLAIN's program writer was limited to goal/subgoal refinement. If no principle could be found to refine a goal, the system could go no further. This limitation reduced the reusability of the system's problem solving knowledge in new situations. As described below, the program writer in EES will be capable of reformulating a goal when a match cannot be found. This ability to reformulate goals will also result in a more explicit development history which we expect will be useful in producing better explanations.

XPLAIN's explanation generator could answer only a few types of questions. Answers were generated by relatively fixed procedures that were coded into XPLAIN itself. In our current work, we are viewing explanation production as a planning task, and expressing knowledge about explanation as a set of declarative explanation strategies. Our desire to answer a broader range of questions has forced us to represent additional kinds of knowledge in the knowledge base of EES.

The primary goal of the EES project is to provide a framework that will facilitate construction of expert systems according to the paradigm of separate models and recorded developments. This requires pursuing the lessons of the XPLAIN system by extending the forms of knowledge known, extending the kinds of development recorded, and extending the benefits beyond explanation to development and maintenance.

III. Design of the EES System

In response to the considerations laid out in the preceding section, we are currently designing and implementing a prototype of the EES framework, which will be used to implement the Program Enhancement Advisor. A broad view of the EES system is shown in Fig. 1.

The knowledge base is the foundation stone of the EES system. The *domain model* describes how the domain works. It contains, among other things, typological and causal linkages. While the domain model describes how the domain works, it does not indicate how problem solving should be done. *Domain principles* represent problem-solving strategies and are used by the program writer to drive the refinement process. *Tradeoffs* are associated with domain principles to indicate the beneficial and harmful effects of selecting a particular strategy to achieve a goal. *Preferences* are associated with goals and are used to set priorities based on tradeoffs. Mappings between abstract terms and the concepts that realize them (which had been included as part of domain principles in XPLAIN), are broken out as a separate type of knowledge to allow *terminology* to be shared across domain principles. *Integration knowledge* is used to resolve potential conflicts among knowledge sources. For example, in the digitalis domain, dosage reduction recommendations based on patients' serum calcium levels have to be integrated with recommendations based on serum potassium levels in order to compute a single overall dosage recommendation. *Optimization knowledge* represents ways of efficiently controlling the execution of the derived expert system.

The EES framework will generate a runnable expert system by applying the *program writer* component to the knowledge base. The steps taken in producing code are recorded in a *development history* (or "*refinement structure*"), a lattice structure whose leaf nodes represent system implementation code and whose interior nodes represent goals and decisions made on the way to generating the implementation. The *interpreter* executes the leaf nodes of the development history. It produces an *execution trace*, and manages the system's normal interaction with users. User questions, however, are sent to the *explanation generator*, which accesses the knowledge base, development history, interpreter, and execution trace in the course of constructing answers to queries.

The program writer consists of two major components. Conceptually, it is simplest to think of them as applying in multiple passes, although they may be interleaved in the actual implementation. The first component makes use of the domain model, domain principles, terminology, inte-

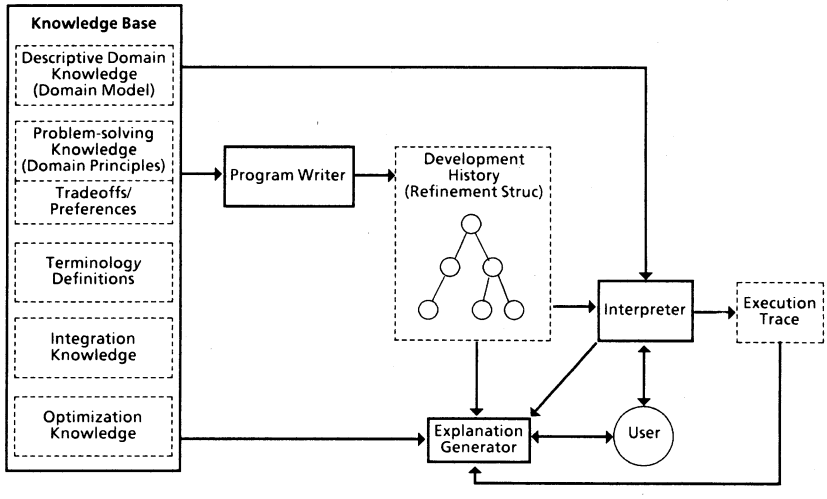


Fig. 1. Global view of the EES framework.

gration knowledge, and preference and tradeoff knowledge to produce an initial version of the expert system. During the second pass, the refinement structure is mod ified by an efficiency optimizer that attempts to use optimization knowledge to find restructurings of the development structure that reduce execution costs.

The remainder of this section will illustrate the concepts introduced thus far by considering the development of a portion of the Program Enhancement Advisor. This portion seeks to improve the readability and maintainability of programs by recoding conditional expressions currently expressed using Lisp's COND into expressions that use higher-level constructs like the IF-THEN-ELSE available in Interlisp's "Conversational LISP" package [5]. We have deliberately selected a transformation whose appropriateness depends on user preferences so that we can consider some issues that arise in such cases. We will first describe the knowledge base related to this example, then consider the process by which the program writer uses the knowledge base to produce executable code.

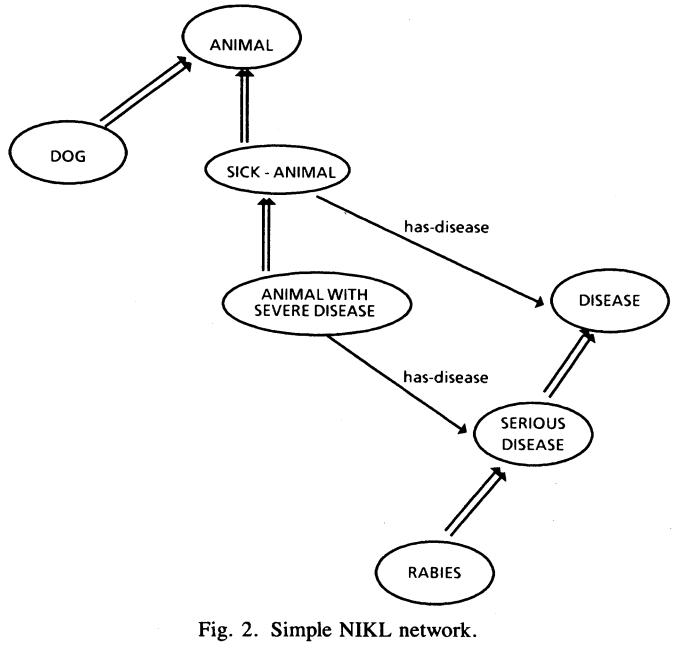


Fig. 2. Simple NIKL network.

A. The Knowledge Base

The system's knowledge base is represented in NIKL [6]. NIKL is a refinement of KL-ONE [7], a semantic net work-based representational formalism. NIKL has the usual complement of representational devices, such as concepts, roles (or slots) on concepts, links between concepts indicating subsumption, and the like. For our purposes, the most attractive feature of NIKL is that the semantics of these devices have been worked out sufficiently so that automatic classification [3] is possible. Given an existing network and a new concept, the NIKL classifier automatically determines the appropriate place for the new concept in the subsumption hierarchy of the network, *based solely on the structure of that concept*. As we will see, this is a very useful feature for knowledge acquisition, plan finding, and explanation.

To see just how this classification operation takes place, consider the sample network in Fig. 2. In this simple network, double arrows indicate a-kind-of (superc) relations, while single arrows indicate attributes (roles) of concepts. Roles point to concepts that indicate the type of filler required for that role. These type restrictions are called *value restrictions*. The roles associated with a concept are definitional—they define what the concept means. Thus, in Fig. 2, a SICK-ANIMAL is defined to be an animal with a has-disease role that must be filled by a DISEASE, while an ANIMAL-WITH-SEVERE-DISEASE has its has-disease role filled by a SERIOUS-DISEASE. Suppose we now define a new concept, MAD-DOG, as shown in Fig. 3. Based on just the information given in Fig. 3 the classifier can infer that a MAD-DOG is a kind of ANIMAL-WITH-SEVERE-DISEASE because it has RABIES which is a SERIOUS-DISEASE, and add the superc link as shown in Fig. 4. Suppose we now further define a new term, RABID-ANIMAL, as shown in Fig. 5. Based solely on the structure of the new concept, the classifier can determine that the concept can be moved down to a more specific position in the network, as shown in Fig. 6. This classification is possible since a RABID-ANIMAL by definition has-disease RABIES and therefore must be a kind of ANIMAL-WITH-SEVERE-DISEASE since such animals by definition have a SERIOUS-DISEASE and RABIES is a SERIOUS-DISEASE.

From the standpoint of EES, the use of NIKL provides

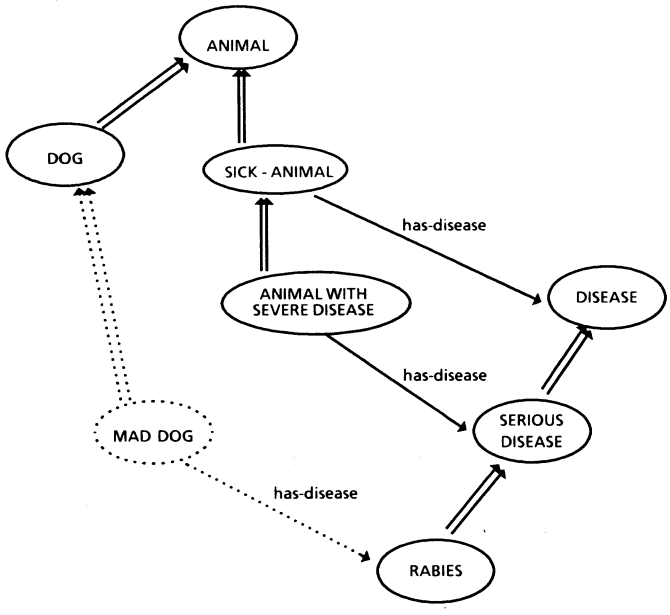


Fig. 3. Defining a new term.

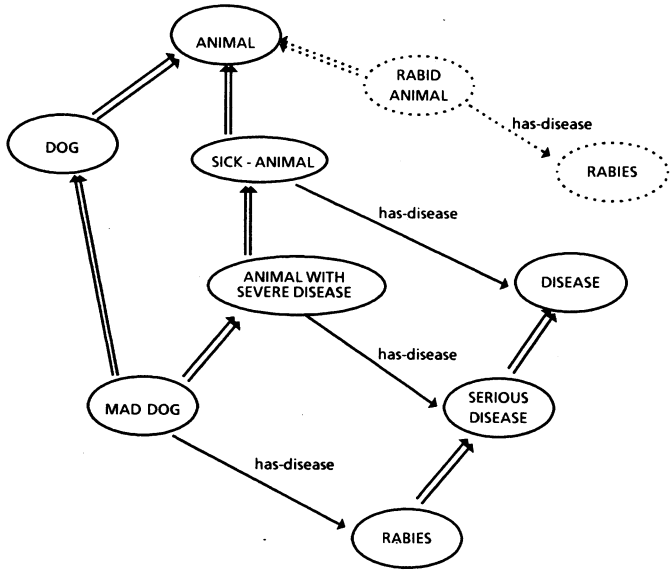


Fig. 5. Defining another term.

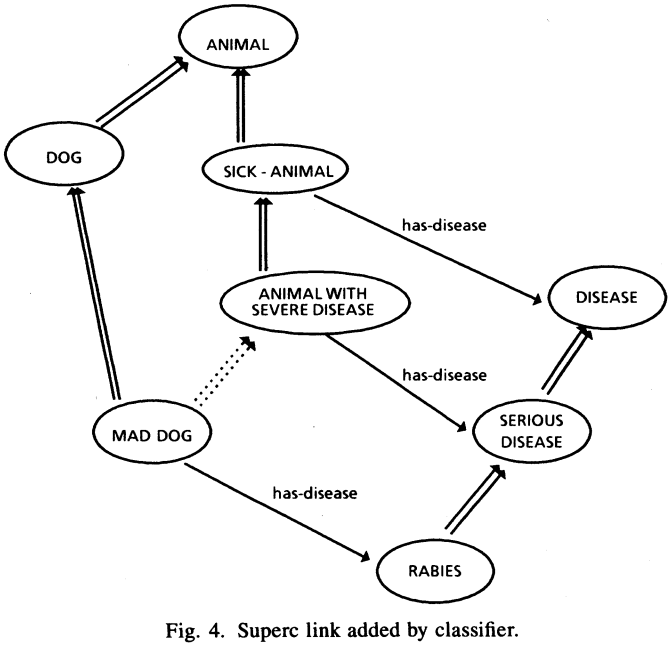


Fig. 4. Superc link added by classifier.

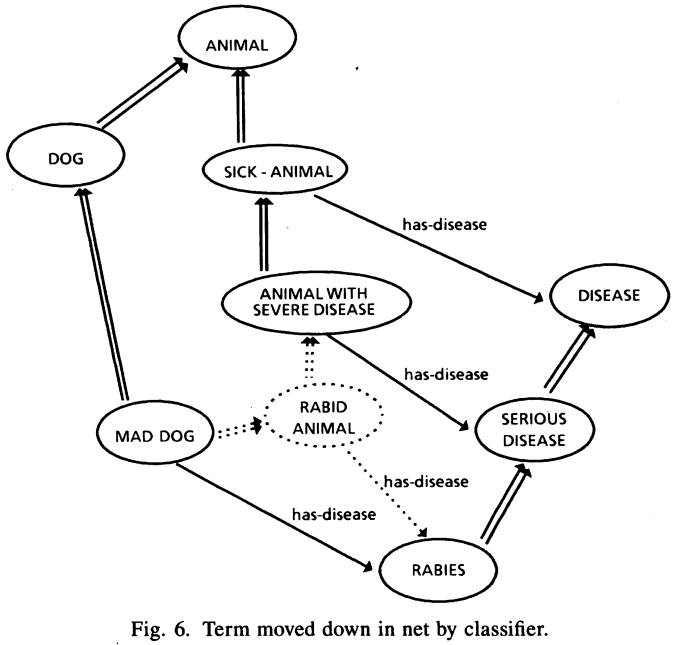


Fig. 6. Term moved down in net by classifier.

us with two valuable features. First, it allows us to give explicit definitions for terminology, overcoming one of the limitations of XPLAIN. Second, we have found it to be very useful to regard the classifier as a sort of pattern-matcher. As such, it is useful both for finding specific instances that match more general terms and, since all of the system's knowledge about plans and goals is expressed in NIKL, the classifier organizes the system's problem-solving knowledge into a hierarchy which is very useful in finding plans for achieving particular goals.

As was described earlier, the knowledge base is the repository for several different kinds of knowledge which are integrated together by the program writer to produce a working expert system. In the knowledge domain of Lisp program enhancement, the predominant domain descriptive knowledge involves the properties of language constructs and transformations between alternative constructs. Problem-solving knowledge centers around using transformations to improve a program. Tradeoff/preference knowledge defines when transformations are considered improvements and provides rules for resolving conflicts when alternative transformations are applicable. Terminological knowledge allows the transformations and problem-solving knowledge to be expressed in high-level terms. Integration and optimization knowledge primarily serve to guide the program writer in generating a runnable implementation of the system by indicating how to coordinate applicability testing across the assorted transformations.

In the Program Enhancement Advisor domain, the *domain knowledge*, or descriptive knowledge of how the domain works, is the knowledge of program transformations: what their applicability criteria are and what effect they will have. A simplified portion of the NIKL representation describing the transformation of a **COND** statement into a CLISP **IF-THEN-ELSE** statement is shown in Fig. 7. It is not necessary to understand the details of

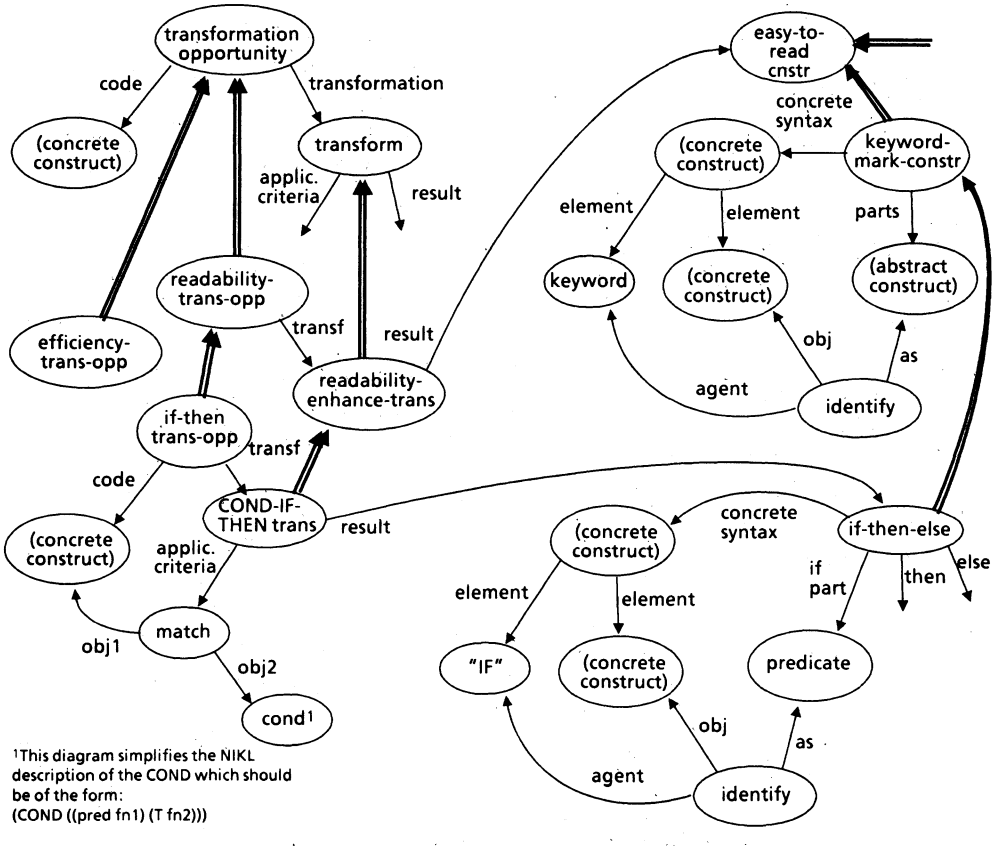


Fig. 7. Simplified NIKL representation of **IF-THEN-ELSE** transform.

the NIKL representation (and space limitations preclude describing them). It is sufficient to point out that the **COND** to **IF-THEN-ELSE** transformation is classified as a kind of readability-enhancing transformation because the result of the transformation is an easy to read construct. This classification follows because the **IF-THEN-ELSE** construct is a kind of keyword-marked construct which, in turn, is an easy to read construct because the parts of the construct are explicitly identified by keywords. In summary, this portion of the net expresses the idea that an **IF-THEN-ELSE** construct is easy to read because each of the components of the construct is explicitly identified by a keyword. (In Section IV we will consider how the system might mechanically create an explanation justifying the desirability of this transformation.)

The *problem-solving knowledge* in the Program Enhancement Advisor is knowledge that tells the system how to use its knowledge of transformations to enhance a program. In particular, this includes strategies for scanning a program file to find places where transformations might be applied, for resolving conflicts among those possible transformation applications, and for finally applying the transformations. Plans and goals are represented in NIKL and are organized into a hierarchy by the NIKL classifier. Associated with each plan is a *capability description* which describes what the plan can do. This description is used by the system to find plans that can achieve goals.

The explicit representation of *terminological knowledge*, or the knowledge of how terms are defined and differentiated, is considerably facilitated by our use of NIKL, because it is exactly the kind of knowledge that has to be represented for the NIKL classifier to do its job. For example, in Fig. 7, a keyword-marked construct is defined as an abstract construct whose concrete syntax has keywords that identify parts of the concrete syntax as components of the abstract construct. This structural description is used by the classifier (and the program writer as described below) to find particular instances of keyword-marked constructs.[3]

B. The Refinement Process

In XPLAIN, the program writer created the expert system in a top-down fashion, starting from a high-level goal that was an abstract statement of what the expert system was intended to accomplish. As the writer implemented goals, subgoals were raised which in turn required implementation. The writer iteratively implemented goals using goal/subgoal refinement until the level of system primitives was reached. This is the familiar form of refinement by breaking a goal down into subgoals. This occurred whenever the system could find a plan that implemented a goal. The system located plans by searching up the classification hierarchy starting from the goal until it found a plan.

In EES, we have retained the goal/subgoal refinement process of XPLAIN, but we have also increased the power of the program writer so that it is capable of reformulating

[3]It is worth pointing out that in the XPLAIN system, because its knowledge base did not support definition of terms and classification, terminological knowledge was represented implicitly in the domain rationale, as part of the domain principles. We now feel that this mixing of terminology with problem-solving knowledge was inappropriate. Terminology should be defined separately so that it can be consistent across domain principles.

a goal automatically if no direct implementation can be found for that goal.

We have identified three different classes of reformulations that seem to be useful in expert system construction. The first reformulation we refer to as an "*and*" reformulation because it involves splitting a goal into parts, each of which must be performed to accomplish the original goal. Integration knowledge must be used to integrate the results returned by each of the pieces into an acceptable result for the original goal. We refer to second and third reformulations as "*or*" reformulations because they involve splitting a goal into parts, of which only one needs to be performed to accomplish the original goal. In performing these reformulations, the program writer must create a selection test that will indicate, at runtime, which subplan should be executed, based on the particular situation at hand. Integration is not necessary for "*or*" reformulations.

Useful reformulations we have identified are as follows.

Covering Reformulation: When the system is unable to find a plan that implements a goal, it may reformulate that goal into several goals that can be implemented and together cover the possibilities presented by the original goal. We feel that this kind of reformulation caused by implementation concerns occurs quite frequently in expert system design. For example, one may want an overall assessment of the symptoms of a disease, but only be able to ask about individual symptoms. This requires splitting the goal into cases and asking about each symptom in turn. Similarly, the Digitalis Advisor had the goal of adjusting the drug dose based on the patient's sensitivities, but it had no direct way of assessing the patient's sensitivities as a collection, so again the goal had to be split into individual cases. Below, we will also illustrate the use of this kind of reformulation in the Program Enhancement Advisor. With the capability of covering reformulations comes the need to be able to recombine the results of individual cases into an overall result for the general goal. For example, once a patient's individual symptoms have been assessed there is still the issue of how to combine those assessments into a reasonable overall assessment of the disease. This is where the system's integration knowledge comes into play. An example of its use in the Program Enhancement Advisor is given below.

User Directed Dynamic Refinement: Most current expert systems do not accept much direction from the user. Yet as expert systems move into domains where the goals are less clear cut, it becomes more important to allow the user to further specify goals. For example, in the Program Enhancement Advisor, the top-level goal of enhancing a program is under-specified. It could be that to enhance a program means to make it more readable, or it could mean to make it more efficient or maintainable. Exactly what is appropriate depends on knowledge that is outside the scope of the Program Enhancement Advisor, so it makes sense to get advice from the user to further specify such goals. However, the system must constrain the user's ability to refine goals lest he push the system beyond its capabilities. We illustrate our approach to providing the user with a constrained ability to specify goals in the example below.

Input-Based Reformulation: Sometimes a goal may be posted that has such a general input that none of the available plans can handle it. However, if a set of plans can be found that accept inputs that together cover the space of possible inputs that the original goal may present, then the original goal may be implemented by dispatching at runtime to one of the more specialized plans, depending on what the input actually is. For an example of the result of such a reformulation, consider the "generic" arithmetic operators provided by many programming languages. These allow a programmer to perform operations without worrying about the types of the arguments. At runtime, a dispatch is made to the appropriate routine based on the types of the actual arguments.

From the standpoint of system construction, providing these reformulations is important because they give the program writer more flexibility. The system builder does not have to worry as much about the exact match up of goals and plans. Instead, he can state what a goal requires or a plan is intended to accomplish and the writer can help with the details of matching the two up. This flexibility in matching can also help to make problem-solving knowledge developed for one expert system reusable in another. Providing these reformulations has only become possible by moving to a richer knowledge representation language such as NIKL, where it is possible to indicate "covering" relationships among concepts.

From the standpoint of explanation, it is important to distinguish each of these means of implementing goals, and to record their use. *Goal/Subgoal refinements* indicate to the explanation facility how a low-level goal fits into the overall strategy expressed by a higher-level goal. Modeling *covering reformulations* explicitly is important because knowing that a particular goal was created due to implementation concerns usually means that that goal is unlikely to be interesting to users (but possibly quite important to system designers). The explicit modeling of user preferences afforded by *user directed dynamic refinement* allows the system to tailor its explanations based on known user desires. For example, if the system knows that the user is just interested in enhancements that improve the readability of a program and the system is asked to justify its recommendation of a transform that enhances both readability and efficiency, it could produce a focused explanation that stressed just readability. Finally, by modeling *input-based reformulation*, the explanation facility can explain to a user that the reason why it used two different methods on different occasions to solve what appeared to be the same problem was that the particular inputs provided caused the system to pick the most appropriate method.

C. An Example

This section outlines a portion of the steps the program writer goes through in generating the Program Enhancement Advisor. Starting from the abstract goal of enhancing a program we will show how the system moves toward

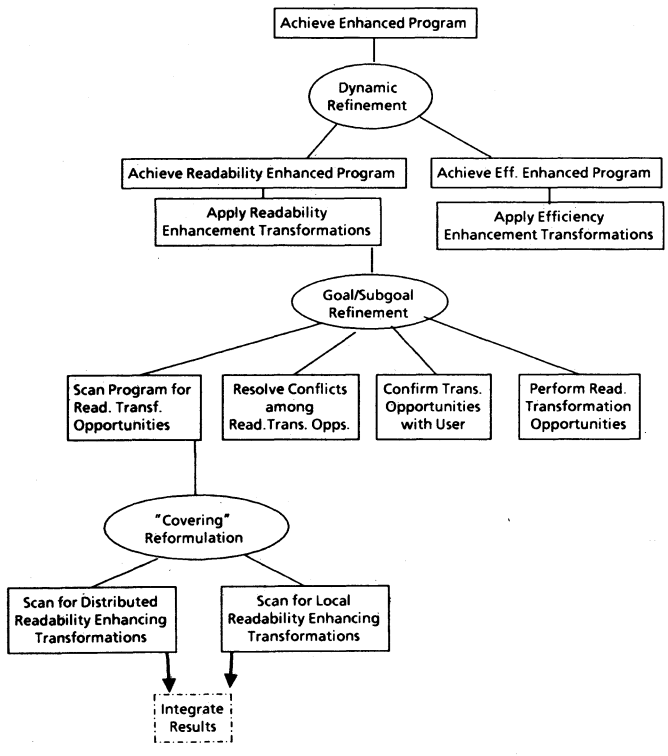
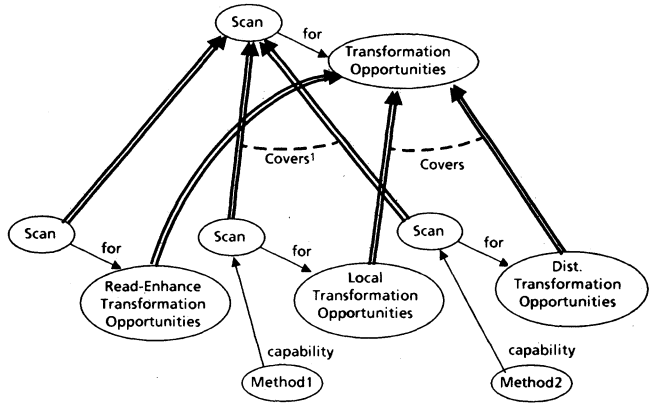


Fig. 8. A simplified portion of the development history.



[1]This coverage relation is derived by the system from the coverage relation between distributed and local transformation opportunities.

Fig. 9. A portion of the method hierarchy.

generating code to scan for specific transformation opportunities. Fig. 8 shows the development history that results from the implementation steps described below.

The system starts with the goal ACHIEVE ENHANCED PROGRAM. A further specification (not shown in the figure) tells the program writer that code should be created to allow the user to specify *at runtime* what kind of enhancement should be performed. Since the program writer cannot predict which kinds the user will request, it will plan code to cover all kinds present in the knowledge base. Suppose it finds two kinds of enhancements, those for enhancing efficiency and those for enhancing readability. The writer would post the goals ACHIEVE READABILITY ENHANCED PROGRAM and ACHIEVE EFFICIENCY ENHANCED PROGRAM as goals to be implemented. It would also create code for interrogating the user and invoking the appropriate subgoal based on the user's desires.

In EES, we have found it useful to distinguish two types of goals: *state goals* that specify a particular state to be achieved and *action goals* that are used to realize the state goals. To achieve the state goal of ACHIEVE READABILITY ENHANCED PROGRAM the system finds a plan that has a single action subgoal APPLY READABILITY ENHANCEMENT TRANSFORMATIONS. That is, a readability enhanced program is being achieved by applying readability enhancement transformations.

Next, the system finds a four-step plan to implement the goal APPLY READABILITY ENHANCEMENT TRANSFORMATIONS (this same plan is also used to implement the goal APPLY EFFICIENCY ENHANCEMENT TRANSFORMATIONS).

1) Scan the program for transformation opportunities.

2) Resolve conflicts among transformation opportunities.

3) Confirm transformation opportunities with the user (to assure that they are acceptable).

4) Perform the transformation opportunities approved by the user.

In the development history, the writer records the implementation of the ACHIEVE ENHANCED PROGRAM goal as a goal/subgoal refinement and specializes the general steps of the plan to reflect the particular goal being implemented (see Fig. 8).

Let us consider how the writer might further refine the goal of scanning for readability enhancing transformations. There is no direct method for implementing this goal, so the program writer examines higher methods in the hierarchy. At the level of SCAN PROGRAM FOR TRANSFORMATION OPPORTUNITIES, the system finds two subconcepts that have specialized methods associated with them (see Fig. 9). One of these scans for local transformations, that is, transformations like the **COND = = > IF-THEN-ELSE** transform where the applicability criteria of the transform can all be verified within a single s-expression. The other scanning method scans for what we call distributed transformations, where the applicability critera require looking at several places in the program. An example of the second type of transform would be one that verifies that it is possible to use records to replace explicit accessor functions. By examining its terminological knowledge, as expressed in NIKL, the writer determines that together these two methods cover the space of possible transforms, so the single goal of scanning for readability enhancing transforms can be re-expressed as the two goals of SCAN FOR DISTRIBUTED READABILITY-ENHANCING TRANSFORMS and SCAN FOR LOCAL READABILITY-ENHANCING TRANSFORMS. This covering reformulation is an instance of a goal being changed based on available domain techniques.

The system continues on in this fashion, refining general goals into increasingly more specific goals, until eventually the level of system primitives is reached. At that point the expert system is complete.

D. Control Issues

How would this approach be used to build expert systems with particular control structures, such as blackboard or backward-chaining architectures? We view this

as a problem of specifying the interpreter that will execute code produced by the program writer. We evaluate three possible approaches below.

Our current approach is to keep the expert system interpreter very simple, and explicitly express the architecture for the system in domain principles. The program writer compiles these principles into a program simple enough for the interpreter to handle. For example, if we wanted a system to perform diagnosis using backward-chaining, we would write a principle that would say, in essence, "To determine whether a physiological state exists, conclude that it does if sufficient evidence for the physiological state exists.– The causal and associational relations that would determine what constitutes evidence for a physiological state would be expressed as domain descriptive knowledge, and integration knowledge would be used to integrate the results of multiple sources of evidence.

This approach seems to be the most appropriate one for the two application domains we have considered, digitalis therapy and program enhancement. It allows us to cleanly and easily intermix different control strategies. Also, because the interpreter is very simple, any sophisticated features of the expert system-s architecture have to be explicitly derived in the development history, so that they can be explained. The major disadvantages are that it may result in an enormous development history, and that the program writer may not be sufficiently powerful to perform all of the derivation steps.

A second alternative would be to raise the level of the interpreter so that the system primitives captured the desired architecture. The program writer would create code for this architecture. The advantages and disadvantages of this approach are just the reverse of the first approach. We have not explored this approach; we prefer the explanatory benefits of the first approach.

The most desirable (but also most difficult) alternative would have the program writer create both a high-level interpreter from simple primitives and the code to run at that high level. A higher-level interpreter would allow the development history to be smaller, and because the interpreter would be explicitly derived, its operation would be explainable. We have not yet explored this approach in detail.

E. Toward Easier Maintenance of Expert Systems

We close this section with a brief discussion of the main advantages that this approach may provide for maintenance and evolution of expert systems:

- *High-level strategies are easily expressed and their refinement is recorded.* In conventional rule-based systems, high-level strategies cannot be easily represented. However, most large rule-based systems are based on some high-level strategies, even though they are only implicitly expressed [8]. These strategies are very difficult to understand just from examination of their rules, but successful modification of the expert system often depends critically upon them. The EES approach makes the strategies, and the record of their refinement into lower-level strategies, explicit. This may give system builders a better chance to understand the workings of an expert system.
- *Separation of different kinds of knowledge makes the system more modular.* By separating different kinds of knowledge, such as problem-solving and descriptive knowledge, each is made more independent of the other. Hence, it is possible to modify one without having to modify the other. This is usually not the case in traditional expert systems where many kinds of knowledge are confounded within a simple rule formalism.
- *Automatic classification of new knowledge eases system modification.* In the example just presented, the NIKL classification structure was used to select transformations based on their enhancement effects, and, having selected such transformations, to find methods for scanning programs for opportunities to use them based on their applicability criteria. Because NIKL supports automatic classification, new knowledge, such as transformations or problem-solving strategies, can automatically be placed in the classification hierarchy and thus be employed consistently and appropriately.

We feel that, together, these features will significantly facilitate the construction of explainable, evolvable expert systems. The approach puts more emphasis on the development phase and will require tools to support construction of richer knowledge representations (see Section V). However, we believe that the costs of extra attention to the development phase will be repaid by several gains, particularly smoother maintenance and the richer systems that might result from a more disciplined approach to specifying their contents.

IV. Explanation—A Process Model for Question-Answering

In previous sections we have described the types of knowledge included in the EES architecture and discussed how this knowledge is organized. One of the primary motivations behind the EES architecture was a desire to provide richer explanations to a broad range of questions. In this section, we will describe the classes of questions we believe are important and discuss how the knowledge available within the EES framework could enable us to provide answers to these questions.

Once we have presented the range of questions we wish to address, we will discuss the issues that arise in actually producing explanations from the knowledge provided in our system. We describe how our explanation strategies are organized, and then demonstrate how our architecture aids in providing a broader range of explanatory capabilities by presenting a detailed example.

A. Answering a Broader Range of Questions

In expert systems which record only the program code, and not the knowledge and reasoning required to generate that code, explanation is necessarily limited to answering questions which depend only on access to that code. Among such questions are primarily questions about *behavior*, such as:

- *How does/did the system perform <action>?*
- *How is/was <parameter> used?*

- *What would be the result of <parameter setting>?*

The explanation capabilities of the XPLAIN system went beyond those of previous systems. XPLAIN could justify why a question was being asked and could answer questions about general as well as specific problem-solving knowledge. However, there are a number of other kinds of questions that might reasonably be asked by a system builder or user. A more complete list of questions follows.

1) Questions of Justification: Questions of justification include questions such as:

- *Why is the system concerned with <parameter, goal, or action>?*
- *Why is <goal or action> necessary (desirable, or important)?*
- *Why should <recommendation> be followed?*

These questions all essentially seek information about the purpose underlying some aspect of the system, that is, about the relationship of that aspect to the goals of the system builder or user. Answering them involves looking at the development history to determine the domain principle(s) that generated the queried object, and from there finding further information by examining related terminology, tradeoffs, and the preferences operative at the point in time under consideration.

2) Questions of Timing or Appropriateness: Both in debugging a system, and in deciding how much faith to put in its conclusions, it is often useful to obtain information about the course of its reasoning processes. This involves questions such as:

- *When did the system consider/reject <goal, action, or conclusion>?*
- *Why did it consider/reject <goal, action, or conclusion> at <time reference>?*
- *Why didn't it consider/reject <goal, action, or conclusion> at <time reference>?*

At one level, these are simply questions about the execution history of a system. Treated as such, they can be answered by techniques such as those in Davis' Teiresias system [9] that recorded triggering conditions for rules, and determined absences that prevented near-miss rules from being satisfied. However, at a higher level, the questions again deal with intentions, i.e., the reasons behind the selection conditions imposed on various knowledge items. Answering such questions may be essentially the same process as justification questions, but it may also tap knowledge that went into deriving the control aspects of the system. This entails examining the development history in search of tradeoffs and preferences that applied in generating choice constraints, and optimization knowledge that was also used to edit the originally planned program.

3) Questions of Definition or Function: These are questions that concern the meaning of a particular concept in the system.

- *What does <term> mean?*
- *What is a <term>?*
- *What are the effects of <action>?*
- *What is the relationship between <term, parameter, goal, or action> and <term, parameter, goal, or action>?*
- *What is the difference between <term, parameter, goal, or action> and <term, parameter, goal, or action>?*

These are essentially questions that involve paraphrasing either the development history, the domain model, or domain principles. In each of these cases, paraphrasing depends on tapping knowledge about terminology.

4) Questions of Capabilities: Both for users to determine when and how much to trust the system, and for system builders to determine where augmentation is required, it is useful to be informed about the abilities and limitations of a system. This involves questions such as:

- *What does the system know about <concept>?*
- *What factors does the system consider/ignore in concluding <conclusion>?*
- *What methods does the system use/avoid in achieving <goal>?*

Questions of this class are particularly likely to be stimulated by answers to previous questions. For example, in the case of the XPLAIN digitalis advisor, the answer to a justification question was that the system was interested in serum calcium levels in order to reduce the recommended dosage if the level was abnormal. This naturally leads to the question, "are there any other factors like serum calcium?" Answering such questions primarily involves searching through the domain model, examining type and causal linkages. However, answering ignore/avoid questions entails looking at the development history to identify processes that have been masked out by the introduction of efficiency optimizations.

B. Producing an Explanation

There are several issues that must be addressed when attempting to provide reasonable explanations to users of an expert system. We believe that our architecture allows us to address these issues in a more satisfactory way than existing expert systems. When generating explanations, one is faced with the problem of deciding what information to include in the explanation and how to express this information. One issue that arises when deciding what information should be included in the explanation is choosing the appropriate level of detail. This issue comes up in several aspects of explanation. For example, when describing the system's behavior to end users, the explanation should focus on goal-based concerns and should not include implementation details. Other systems have addressed this issue by marking certain portions of the system's code as implementation specific, thus indicating that these portions of the code should not be included in explanations [10], [11].

In our system, the explanation generator knows about certain structures in the development history which allow it to determine which goals are generated as a result of implementation concerns and which are problem-solving goals appropriate for inclusion in explanations to end users. We will see an example of this later.

C. Question Types and Answering Strategies

In order to devise a process model for answering the range of questions discussed above, we found it useful to categorize the questions we wished to answer into several *question types*. Based on the question type, the explanation facility selects a strategy to be used in answering the question. Other researchers in the area of question-answering have found it useful to identify question categories and organize procedures for answering questions around these types [10], [12], [13].

A review of the strategies associated with each question type is beyond the scope of this paper, as is a discussion of the interface which allows users to access question-answering capabilities. Here we will concentrate on a discussion of one question type and the strategy for answering it.

D. An Example

In this section we present a detailed examination of the strategy used in answering questions of the type *Justify Recommendation*. Questions which fall into this category are looking for justification of a method or recommendation in terms of the user's goals, the user's stated preferences, or the system's knowledge about trade-offs. For example, "*Why should the <recommendation> be followed?*"

Part of the question analysis process will categorize the question as one of the question types defined in the system. Associated with the question type is the strategy to be used in generating an answer to questions of this type. Here we present a simplified version of the strategy to be used in answering questions categorized as type *Justify Recommendation*.[4]

1) Search the development history for the <method> that produced <recommendation>.

2) Search upward through the development history for the <goal> that this <method> is a plan for achieving. Continue searching upward until reaching a goal that the user shares. (The user is assumed to share the top-level goals of the system.)

3) State this <goal>.

4) State general <method> that is used to achieve <goal>.

5) State how <recommendation> is involved in achieving <goal>.

Now, we will show how this strategy is used to generate a response to an example question of type *Justify Recommendation*. Suppose that the system had just presented the following recommendation:

The construct:

```
(COND ((ATOMP X) (LIST X))
      (T X))
```

may be replaced by the construct:

```
(IF (ATOMP X)
    THEN (LIST X)
    ELSE X)
```

Further suppose that the user then asks why he should apply this particular transformation. Once the user's question has been recognized as being of type *Justify Recommendation*, the system will apply the strategy described above to produce the following explanation:

The system is trying to enhance the readability of the program by applying readability enhancing transformations. **COND** *to* **IF-THEN-ELSE** *is a readability enhancing transformation because* **IF-THEN-ELSE** *has keywords which identify its abstract components.*

In order to generate the first sentence, the system must perform the first two steps of the strategy shown above. It begins by searching through the development history looking for the <method> that produced <recommendation>. (At this point <recommendation> is instantiated to: APPLY COND TO IF-THEN-ELSE TRANSFORMATION.) Referring to Fig. 8, we see that the <method> which produced <recommendation> is the one which scans over s-expressions to check if a transformation like COND TO IF-THEN-ELSE is applicable, and, if so, adds it to the list of enhancement opportunities.

Next, we search upward through the development history for the <goal> which this <method> is a plan for achieving. In performing this search, we skip over those goals which are results of the program writer's implementation concerns. Such goals frequently arise when the program writer must reformulate a single goal into several cases which together cover this goal (see Section III-C on goal reformulation into cases). The development history records which goals were generated due to implementation concerns (see Fig. 8). Thus the explainer can determine which goals must be skipped when looking for a goal that is appropriate to incorporate into an explanation to an end user. The result of the search upward in the development history is the goal ACHIEVE READABILITY ENHANCED PROGRAM. Note that we do not have to go all the way up to the goal ACHIEVE ENHANCED PROGRAM because we assume that the user has provided input at the level of the dynamic refinement node. Thus we can assume that he knows that the goal ACHIEVE READABILITY ENHANCED PROGRAM is a refinement of the goal to ACHIEVE ENHANCED PROGRAM.

Next, we state the goal ACHIEVE READABILITY ENHANCED PROGRAM and the general method which is used by the system to achieve this goal. This method is APPLY READABILITY ENHANCING TRANSFORMATIONS in Fig. 8. At this point, we have generated the first sentence of the explanation above. Besides the lexical knowledge of how to express the goal and method we have stated, the explainer needs to have the knowledge that goals are satisfied "by" methods.

Finally, we use the domain model in conjunction with the NIKL classifier to describe how <recommendation>, in this case APPLY COND TO IF-THEN-ELSE TRANSFORMATION, is involved in achieving the goal ACHIEVE READABILITY ENHANCED PROGRAM. In this case <recommendation> is an instance of a method for achieving the goal. Thus we will describe how <recommendation> qualifies as an instance of the method APPLY READABILITY ENHANCING TRANSFORMATION. This requires stating why the particular **COND** to **IF-THEN-ELSE** transformation recommended is an instance of a readability enhancing

[4]This strategy considers only goals. It does not take into account any preference or tradeoff knowledge.

transformation. The NIKL classifier is used to provide this information. Looking at the domain model (see Fig. 7), we see that the distinguishing characteristic of a readability enhancing transformation is that its result role is filled by an easy-to-read construct. Thus, when attempting to classify a concept as a readability enhancing transformation, the classifier must verify that its result may be classified as an easy-to-read construct. When queried about the recommendation, a particular **COND** to **IF-THEN-ELSE** transformation, the classifier will return information stating how the result of this **COND** to **IF-THEN-ELSE** transformation, namely a particular **IF-THEN-ELSE** construct, is classified as an easy-to-read construct. This will include information about all of the subsumption relationships that hold between the **IF-THEN-ELSE** construct being classified and the concept easy-to-read construct. See Fig. 7. This includes the information that a particular **IF-THEN-ELSE** construct classifies under the generic **IF-THEN-ELSE** definition which in turn classifies as a keyword-marked construct. In addition a keyword-marked construct classifies as an easy-to-read construct. The information returned from the classifier also notes that IF, THEN, and ELSE all classify as keywords, that predicate classifies as an abstract construct, etc.

At this point, we wish to describe the essential characteristics of the **IF-THEN-ELSE** construct which allow it to classify as an easy-to-read construct. Although it would not be particularly readable, we could therefore generate an explanation which states that **IF-THEN-ELSE** is an easy-to-read construct because it has a keyword IF which identifies its predicate, and do likewise for the keywords THEN and ELSE. However, since we wish to generate more abstract explanations, we have heuristics in the explainer which note such parallel structure and form generalizations. When we note that **IF**, **THEN**, and **ELSE** are all keywords and that they all participate in identify relations with concepts which are abstract components, we can form the generalization which appears in the last sentence of the example explanation.

Here we have seen an example of an explanation strategy which allows us to answer one type of question. Similar strategies are being developed to answer the other types of questions we described earlier in this section.

V. Directions

A. Current Status

As of this writing, the EES framework is being implemented, as is the Program Enhancement Advisor.

Using programs volunteered by research programmers in our laboratory as input, and relying on both our own Lisp knowledge and the expertise of two highly skilled builders and maintainers of Lisp-based systems with over a dozen years of experience each, we have identified approximately a dozen enhancements that the program ought to be able to perform. Interviews with our domain experts, and analysis of their thinking-out-loud protocols given as they rewrote the example programs, were used to identify useful enhancements and to try to determine the principles underlying them. As can be expected in an approach that relies heavily on detailed knowledge representation, much of the preliminary work goes into identifying key general concepts in the knowledge domain and determining how to model those concepts in the representation language. Most of those policy decisions have now been made. We have developed an initial-pass version of a language for expressing goals and plans, which translates to their underlying NIKL representation. We have set ground rules for representing concepts such as transformations, syntactic structure of programming constructs, semantic components of programming constructs, and some aspects of the effects of evaluating an expression. Using those ground rules, we have completed the process of encoding the domain and problem solving knowledge related to the **COND** = = > **IF-THEN-ELSE** transformation. We are in the process of encoding the knowledge pertaining to three other transformations: rewriting selected **COND**'s as **SELECTQ**'s, eliminating redundant code in conditionals, and replacing ad hoc data structures by **RECORD** declarations.

We have developed a higher-level language of defining forms for concepts, goals, and plans; specifications in this language are translated into definitions in the underlying NIKL knowledge base. The structures in this underlying knowledge base constitute the input for the EES program writer. Implementation of the planning (or first pass) component of the program writer, which generates the initial refinement structure representing the preoptimized implementation, is also almost complete. We expect soon to begin debugging the handshaking between the knowledge base and the program writer.

For the Program Enhancement Advisor domain, the primary optimizations we expect the expert system to need have to do with collapsing redundant scanning performed in checking for the applicability of individual transformations. The design of the portion of the program writer that will handle those optimizations has begun, but is still preliminary.

To create an English paraphraser of NIKL knowledge structures, one of the critical components of the explanation generator, we are generalizing an existing NIKL paraphraser [14] to employ more principled methods for producing natural language. Explanation strategies have been devised for 6 of our 13 question types, but none have yet been implemented.

B. Future Directions

The EES system is intended as a tool to facilitate the production of expert systems. Thus, our users are both system-builders and the end users of the expert systems constructed with EES. It is worthwhile to consider our plans for addressing their needs.

The most critical determinant of a successful system built with EES will be, for some time to come, the skills of the system builder at specifying conceptual models of the domain of expertise. The approach relies heavily on its underlying knowledge representation language. Knowledge engineers using EES need to be highly conversant in both knowledge representation in general, and the underlying language in particular, in addition to having mastered the art of knowledge elicitation. As our higher-level knowledge specification language evolves, the

depth of familiarity knowledge engineers must have about the underlying representation language will be somewhat reduced. Furthermore, the very explanation capabilities that the system is centered around should be a significant aid to knowledge engineers in understanding, extending, and correcting the representations they must build.

Nevertheless, the ease of building expert systems via the EES approach will be heavily dependent on the presence of support tools to aid in the construction, testing, and modification of the knowledge base from which the actual expert system will be derived. A knowledge acquisition aid, called the **Intelligent Agenda Manager**, is being developed to address this need. This aid will assist the builders of knowledge representations in planning their activities and keeping track of their status during the development process. The principle underlying it is that the increased reliance on detailed knowledge representations is not just a problem, it is also an opportunity. Because more concepts are explicitly represented, it becomes easier to communicate about how one intends to make use of them. Thus, the Intelligent Agenda Manager will provide a language in which system-builders can specify *expectations* about the finished form and content of knowledge bases under construction. These expectations primarily consist of generalizations about where particular concepts should be placed in a classification hierarchy, and/or relationships intended to be created between instances of particular classes. The Agenda Manager will maintain a goal tree, or *agenda*, of "unfinished business" for the system builders by two primary processes. First, it will record expectations explicitly stated by the system builder and use meta-knowledge about knowledge representations to help plan how to achieve those goals. Second, it will generate tests for violations and satisfactions of expectations, modifying the agenda appropriately upon detecting that such an event has occurred.

In the near term, responsibility for the quality of a system from the end users' viewpoint will rest primarily with the system builder. The EES approach will provide two gains that the system builder can take advantage of: the ability to generate natural language explanations in response to user questions, and the ability provided by the EES notion of runtime preferences for users to tailor an expert system more specifically to their needs. It will initially be up to the system builder to pass these gains on to end users by building adequate conceptual models and by programming interactions to obtain information about preferences. Preliminary studies of end users interacting with simulated EES products, however, indicate that user interface issues will play a large role in determining whether full advantage is taken of the novel capabilities EES will provide. In the long term, therefore, it would be desirable for EES to take more responsibility for constructing user interfaces, so that less of the effort involved in following useful interface principles has to be duplicated across system builders.

VI. Closing Comments

In the sections above, we have argued for a new paradigm of expert system development, in which deep models separate and explicitly represent the different forms of knowledge that go into the implementation of an expert system, and in which a recorded development history is kept to trace the intertwining of those different forms of knowledge into runnable code. We have considered some particular forms of knowledge and development that seem likely to be important, discussed our design proposal for an EES system that makes use of them, and tried to show how that system might facilitate a broader range of explanations than is possible under current expert system technology.

In addition to its implications for explanation, we believe this approach also offers other benefits related to development and maintenance. Separating the different forms of knowledge reduces the amount that has to be changed when moving to a new domain. The separation, combined with the support of the hierarchical planner, also means that less has to be done when adding new knowledge about a given domain; producing additional code to account for a new concept involves making a few assertions and rerunning the program writer rather than engaging in extensive manual recoding. In addition to these maintenance-related benefits, there are also potential gains in the development process, since the discipline of explicitly specifying domain knowledge and principles is likely to make errors and inconsistencies more readily apparent.

Acknowledgment

We would like to thank B. Balzer and J. Mostow for their comments on previous drafts. We would also like to thank N. Goldman, T. Lipkis, N. Sondheimer, D. Voreck, and D. Wile for helpful input on other occasions.

References

[1] W. Swartout, "XPLAIN: A system for creating and explaining expert consulting systems," *Artificial Intell.*, vol. 21, no. 3, pp. 285–325, Sept. 1983; also available as Rep. ISI/RS-83-4.

[2] P. Szolovits, "Toward more perspicuous expert system organization," in *Rep. Workshop on Automated Explanation Production (SIGART Newlett.)*, W. Swartout, Ed., ACM, 1983.

[3] J. G. Schmolze and T. A. Lipkis, "Classification in the KL-ONE knowledge representation system," in *Proc. 8th IJCAI*, IJCAI, 1983.

[4] E. Sacerdoti, "A structure for plans and behavior," SRI, Tech. Rep. TN-109, 1975.

[5] *Interlisp Reference Manual*, Xerox Corp., 1983.

[6] M. G. Moser, "An overview of NIKL, the new implementation of KL-ONE," in *Research in Natural Language Understanding*, Bolt, Beranek, and Newman, Inc., Cambridge, MA, Tech. Rep. 5421, 1983.

[7] R. Brachman, "A structural paradigm for representing knowledge," Bolt, Beranek, and Newman, Inc., Tech. Rep., 1978.

[8] W. Clancey, "The epistemology of a rule-based expert system—A framework for explanation," *Artificial Intell.*, vol. 20, no. 3, pp. 215–251, 1983.

[9] R. Davis, *Knowledge-Based Systems in Artificial Intelligence*. New York: McGraw-Hill, 1980.

[10] B. G. Buchanan and E. H. Shortliffe, *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project*. Reading, MA: Addison-Wesley, 1984.

[11] W. Swartout, "Producing explanations and justifications of expert consulting systems," Massachusetts Inst. Technol., Cambridge, Tech. Rep. 251, 1981.

[12] W. G. Lehnert, *The Process of Question Answering*. Hillsdale, NJ: Lawrence Erlbaum Associates, 1978.

[13] K. R. McKeown, "Generating natural language text in response to questions about database structure," Ph.D. dissertation, Univ. Pennsylvania, Philadelphia, 1982.

[14] T. Lipkis and W. Mark, "Knowledge base explainer design docu-

ment," USC/Inform. Sci. Inst., CONSUL Project Internal Design Note.

Robert Neches received the Ph.D. degree in psychology from Carnegie-Mellon University, Pittsburgh, PA, in 1981.

He then spent a year as a postdoctoral fellow at the Learning Research and Development Center of the University of Pittsburgh. His research in machine learning produced a system called HPM that emulated developments in children's mathematical skills, and (with P. Langley) a language called PRISM which has been used in a number of machine learning projects. Since joining Information Sciences Institute, Marina del Rey, CA, in 1982, his research and publications have concerned both expert systems and artificial intelligence applications to friendly computing environments.

William R. Swartout received the B.S. degree in mathematical sciences from Stanford University, Stanford, CA, and the M.S. and Ph.D. degrees in computer science from the Massachusetts Institute of Technology, Cambridge.

He is a Research Scientist at the University of Southern California Information Sciences Institute, Marina del Rey, CA, where he was the principal designer of two systems for making program specifications more understandable, the Gist Paraphraser and the Behavior Explainer. He is currently Project Leader of the Explainable Expert Systems project at ISI, developing a framework for creating expert systems that makes them more understandable and maintainable. His research interests include expert systems, natural language generation, and program synthesis.

Johanna D. Moore received the B.S. degree in mathematics/computer science and the M.S. degree from the University of California, Los Angeles, in 1980 and 1982, respectively. Her Master's thesis dealt with the issue of transaction management in a distributed operating system.

She is currently working toward the Ph.D. degree in artificial intelligence at UCLA and the University of Southern California Information Sciences Institute. Her research deals with planning explanations, natural language generation, and user modeling.

The Role of Domain Experience in Software Design

BETH ADELSON AND ELLIOT SOLOWAY

Abstract—**A designer's expertise rests on the knowledge and skills which develop with experience in a domain. As a result, when a designer is designing an object in an unfamiliar domain he will not have the same knowledge and skills available to him as when he is designing an object in a familiar domain. In this paper we look at the software designer's underlying constellation of knowledge and skills, and at the way in which this constellation is dependent upon experience in a domain. What skills drop out, what skills, or interactions of skills come forward as experience with the domain changes? To answer the above question, we studied expert designers in experimentally created design contexts with which they were differentially familiar. In this paper we describe the knowledge and skills we found were central to each of the above contexts and discuss the functional utility of each. In addition to discussing the knowledge and skills we observed in expert designers, we will also compare novice and expert behavior.**

Index Terms—**Artificial intelligence, cognitive models, cognitive science, software design.**

I. Introduction: Motivation and Goals

A DESIGNER'S expertise rests on the knowledge and skills which develop with experience in a domain. As a result, when a designer is designing an object in an unfamiliar domain he will not have the same knowledge and skills available to him as when he is designing an object in a familiar domain. In this paper we look at the software designer's underlying constellation of knowledge and skills, and at the way in which this constellation is dependent upon experience in a domain. What skills drop out, what skills, or interactions of skills come forward as experience with the domain changes? Specifically we ask the following.

- *What happens when well-known information must be used in novel ways?*
- *What happens when domain experience cannot be relied upon? What are the general knowledge and skills which remain?*
- *What happens when the object has been designed before?*

To answer the above questions, we studied expert designers in experimentally created design contexts with which they were differentially familiar. In this paper we describe the knowledge and skills we found were central to each of the above contexts and discuss the *functional utility* of each; what role it played, how it contributed to getting the job done. In addition to discussing the knowledge and skills we observed in expert designers, we will also compare novice and expert behavior. The comparison points out the utility of the knowledge and skills of the expert.

Manuscript recieved June 3, 1985; revised July 8, 1985. This work was supported by the National Science Foundation under Grants IST-8505019 and IST-8519255.

B. Adelson is with the Department of Computer Science, Yale University, New Haven, CT 06520, and the Division of Information Science and Technology, National Science Foundation, Washington, DC 20050.

E. Soloway is with the Department of Computer Science, Yale University, New Haven, CT 06520, and Compu-Teach, Inc., New Haven, CT 06520.

0098/5589/85/1100-1351$01.00 © 1985 IEEE